NIPALS optimization notes

Kevin Wright

October 25, 2017

Abstract

These are some notes to document some of the optimization process for the `nipals` package. As such, it is not complete and the R code chunks cannot be re-run.

Optimizing performance is an art form that requires a good understanding of how functions manage memory and calculations, but also involves a fair bit of trial and error. For example, code that is optimal for small data may not be optimal for large data. There can also be a trade-off between code that is optimal and code that is readable. Our view leans heavily toward the philosophy that programmer time is more expensive than processor time, so code should be written for humans.

General computational performance tips

In this section `x` and `y` are matrices and `v` is a vector.

1. When possible, avoid looping over the columns of a matrix. Instead, use `apply` and similar functions.
2. Do not use `cbind` (or `rbind`) to assemble results into a matrix. Instead, initialize a full matrix of NA values and insert the results into the appropriate column of the matrix.
3. Use `x*x` instead of `x^2`.
4. Use `sqrt(x)` instead of `x^0.5`.
5. Use `crossprod(x,y)` instead of `t(x) %*% y`, since the latter has to transpose first and then multiply.
6. Use `crossprod(v)` instead of `sum(v*v)` if `v` has a lot more than a million numbers or if the result could have numeric overflow.

```
v = rnorm(1e8) # 100 million
system.time(sum(v*v))
#   user  system elapsed
#   0.24    0.17    0.40
system.time(crossprod(v))
#   user  system elapsed
#   0.24    0.00    0.23

v = rnorm(1e9) # 1000 million
system.time(sum(v*v))
#   user  system elapsed
#   3.25   45.71  141.76
system.time(crossprod(v))
#   user  system elapsed
#   2.99    0.72   19.20

v = 1:1e6 # 1 million
system.time(crossprod(v))
#   user  system elapsed
```

```
#      0       0       0
system.time(sum(v*v))
#   user  system elapsed
#      0       0       0
#Warning message:
#In k * k : NAs produced by integer overflow
```

7. Use `colSums(x*x)` instead of `diag(crossprod(x))` if `x` is much wider than 1000 columns.

```
x = matrix(rnorm(10000), nrow=10, ncol=1000)
system.time(colSums(x*x))
#   user  system elapsed
#      0       0       0
system.time(crossprod(x))
#   user  system elapsed
#   0.83    0.14    0.97
```

8. Avoid making copies of data structures, and avoid repeating calculations.

Calculating scores t = Xp/p'p

Part of the NIPALS algorithm involves iterating between calculating the loadings **p** and the scores **t**. This section shows some of the ideas that were tried to increase the performance of the calculation of the **t** vector.

For testing purposes, a 100×100 matrix is big enough so that tweaks to the code will show differences in performance time, but small enough so that each call of the function does not require a lot of waiting. A missing value is inserted to force the function to use the method needed for missing data.

```
set.seed(42)
Bbig <- matrix(rnorm(100*100), nrow=100)
Bbig2 <- Bbig
Bbig2[1,1] <- NA
```

For the optimizing process, we use code taken from `mixOmics::nipals` since it avoids `for` loops over the columns of **X**, and should have better performance than `ade4::nipals` or `plsdepot::nipals`.

The timings are the median of 3 runs. The timings in this section were recorded before the Gram-Schmidt orthogonalization step was added.

Version 1

This is the version taken from the `mixOmics` package.

```
th = x0 %*% ph
P = drop(ph) %o% nr.ones # ph in each column, nr.ones is a vector of 1
P[t(x.miss)] = 0
ph.cross = crossprod(P)
th = th / diag(ph.cross)

system.time(res0 <- nipals(Bbig2, ncomp=100))
#   user  system elapsed
#  10.76    0.00   10.78
```

Version 2

There's no need to store the `ph.cross` object, and the `diag(crossprod())` is needlessly expensive since we only need the diagonal elements. This is an easy change and has a big reward.

```
th = x0 %*% ph
P = drop(ph) %o% nr.ones # ph in each column
P[t(x.miss)] = 0
th = th / colSums(P*P)

system.time(res <- nipals(Bbig2, ncomp=100))
#   user  system elapsed
#    4.4     0.0     4.4

all.equal(res0, res)
# TRUE
```

Version 3

Most of the columns of `P` are the same, so the element-wise multiplication `P*P` is repeating a lot of the same multiplications in the different columns. Better to square the numbers in one column, then put those into all columns. Also, there's no need to calculate `th` in two steps.

```
P2 <- drop(ph*ph) %o% nr.ones # ph in each column
P2[t(x.miss)] <- 0
th = x0 %*% ph / colSums(P2)

system.time(res <- nipals(Bbig2, ncomp=100))
#   user  system elapsed
#      4       0       4

all.equal(res0, res)
# TRUE
```

Version 4

The first line of code is squaring the elements of `ph`, then outer-multiplying by a vector of 1s to insert these into each column of `P2`. It makes sense algebraically, but we can avoid the multiplications and just build the matrix `P2` by recycling the first column.

```
P2 <- matrix(ph*ph, nrow=nc, ncol=nr)
P2[t(x.miss)] <- 0
th = x0 %*% ph / colSums(P2)

system.time(res <- nipals(Bbig2, ncomp=100))
#   user  system elapsed
#   3.38    0.00    3.41

all.equal(res0, res)
# TRUE
```

Comments

Although this look like an easy optimization, there were numerous other (failed) versions, and each version often required fiddling with the syntax to make sure the right results were calculated. For example, what happens if `ph` is a vector (not a matrix) and is put into a matrix operation? Not always what you might expect.

The optimizations described above reduced the user time from 10.76 seconds to 3.38 seconds.

The total effect of all optimizations in the algorithm reduced the user time for the `nipals` function from 19.20 seconds to 3.38 seconds in this example.

Calculating PP' and TT'

In the Gram-Schmidt orthogonalization part of the algorithm, it is necessary to calculate $P_h P_h'$ where P_h is a matrix of the first h columns of the loadings matrix P. It is not necessary to re-calculate the entire $P_h P_h'$ product for each Principal Component, but only to update the product $\mathbf{P_h P_h'} = \mathbf{P_{h-1} P_{h-1}'} + \mathbf{p_h p_h'}$. Here's a numerical illustration:

```
set.seed(42)
P = matrix(rnorm(9), 3)
PPp = P %*% t(P)
all.equal(PPp,
          P[,1,drop=FALSE] %*% t(P[,1,drop=FALSE]) +
          P[,2,drop=FALSE] %*% t(P[,2,drop=FALSE]) +
          P[,3,drop=FALSE] %*% t(P[,3,drop=FALSE]) )
# TRUE

all.equal(PPp,
          tcrossprod(P[,1]) + tcrossprod(P[,2]) + tcrossprod(P[,3]) )
# TRUE

# multiply by a  vector
all.equal( PPp %*% 1:3,
           tcrossprod(PPp, t(1:3)) )
# TRUE
```

Using the 100×100 matrix example, the Gram-Schmidt method adds only a modest increase in time.

```
system.time(m1 <- nipals(Bbig2, ncomp=100, gramschmidt=FALSE))
#   user  system elapsed
#   3.68    0.02    3.70
system.time(m2 <- nipals(Bbig2, ncomp=100, gramschmidt=TRUE))
#   user  system elapsed
#   4.29    0.03    4.37
```

R vs C comment

The `nipals()` function makes heavy use of the `crossprod()` and `tcrossprod()` functions, which are already extensively optimized. Non-optimized coding of the NIPALS algorithm in C would probably be *less* efficient than the R version used in this package.